Directed Study: Use of CNNs for image classification from Sentinel-2 imagery training patches
Professor Millard
December 14th, 2021
Michael

Introduction

Identifying objects from aerial imagery has been a longstanding requirement in many fields, with different applications, e.g., ecology, conservation, and in plenty of other agricultural domains. As times progress, so does the vast amount of imagery sensor data that is openly available, but a problem begins to arise. How can we keep up with the vast torrent of ever-increasing data that exceeds human comprehension? Machine learning (ML) is the answer, ML is a form of artificial intelligence that has been used to train a particular system using provided data. In the confines of remote sensing, ML is primarily used in image classification-based problems, because while there is a finite limit to the number of details the human eye can see, the same cannot be said for machines. In this report, an entire ML workflow using a region of interest (ROI) over Ottawa will be examined and completed to produce classified images along with custom training patches from the Sentinel-2 satellite. The required prerequisite knowledge is discussed in ‘Background’, while ‘Methods’ discusses how the scripts are to be executed, and the ‘Conclusion’ provides a reflection on lessons learned.

Background

Image classification is the task of taking an input image and outputting a perceived class or probability of classes stating the likelihood of what best describes the image [1]. There is a multitude of methodologies and techniques to address this problem, with two of the most popular and commonly used being Pixel based-object classification and Feature based-object recognition. Pixel based-object classification works just as it sounds, by analysing the individual pixels of an image, before compiling a guess of what the object/image most likely is. These classifiers are extremely effective when used with well-defined/separated classes. However, issues begin to arrive in the minute details, or in areas where the spectral information is nigh identical or extremely similar, for example in an urban land use situation; rooftops, car parks, and roadways can all be constructed from asphalt and in the case where they need to be sub-classified or separated, problems will ensue. Similarly, if the quality of the image is low, which is often an issue in remote sensing, the results will likely not be good enough for solid image recognition.

This is where Convolutional Neural Network (CNN) Feature based-object classification comes into play. As the name describes, it tries to extract certain features from an input image to perform classification. You take the input image and pass it through a series of convolutional layers for analysis, before receiving an output [2]. The first layer in a CNN is always a Convolutional Layer, and the input although an image, it’s formatted so a computer can easily read it, meaning the image is converted to a 32 x 32 x 3 array of pixel values. A filter is then applied the input image, usually also in the form of an array of numbers, which are commonly called weights or parameters. The filter must be the same depth as the input image array. As the filter slides or moves around the image (also known as convolving) it is multiplying the values in the filter with the original pixel values of the input image. The area that is currently being filtered is called the receptive field. As the filter completes its multiplication, the final values are summed up into a single number that is only representative of when the filter is convoluted in that location. This process is then repeated for every location of the input volume (array), and each unique location produces a new weight. After finishing sliding over the entire image, a new array of numbers is produced and called an activation map or feature map.

A CNN can have a single or multiple convolutional layers. Each can be thought of as a filter identifier, or things with simplistic characteristics that nearly all images have in common (think lines, edges, primary colors/shades, and bends). For example, if the first filter is a straight-line detector, the filter will have a pixel structure in which higher numerical values will be located along the area that is a linear representation. Next, as the filter slides over the input image, if a shape that generally resembles the line the filter is representing, all multiplication are summed together and will result in a larger value. In the case where there wasn’t anything in the image section that responds to the line detector filter. The corresponding activation map will show area where there is a higher likelihood of some sort of line, whereas the lower values show areas where there is less likely to be lines. To build up an entire image filter, many convolutional layers are layered upon each other, to build up a clear and complete picture of the desired object to be detected.

Generally, other types of layers are spread between the convolutional layers, to both provide preserve dimensionality and improve the overall robustness of the network in addition to preventing overfitting. The last layer in a CNN, known as the ‘Fully Connected layer’, is an amalgamation of all the previously discussed high level features (filters, convolutional layers, windows, etc.) [2]. The purpose of this last fully connected layer is to accept an input volume (last output of the layer preceding it) and output an N dimensional vector, where N is the number of possible classes the program can choose from. Each number within this N dimensional vector, represents the likelihood of a particular class. This likelihood is determined based off the previously mentioned features (activation maps, filters, etc.), that it takes as input, before determining which features most correlate to a particular class. A fully connected layer also usually includes carefully adjusted weights so that when the products between the weights and the previous layer are computed, the correct probabilities for each of the different classes is received [4].

Once a network is built, it must then be trained. This is done via a process called ‘backpropagation’. And unless otherwise specified, a new CNNs weights/filters are generally randomized values that have yet to be tuned [5]. Much like how a newborn baby doesn’t know what a cat, dog, or frog is. However, as time goes on and the child gets older, they are shown different images/objects and the correct corresponding label. This idea of labelling images and learning is essentially the same process the CNN goes through to learn and adjust hyper-parameters during backpropagation. As the parameters are tuned and adjusted the structure of the layers may change or alter slightly. A trained network is almost always then cross-validated to evaluate and test performance to see where (if any) improvements are needed [6].

Uses for fully trained and accurately validated models are nigh limitless in the modern world. In the confines of remote sensing, CNNs have proved to be an invaluable resource thus far in the analysis of earth observation data. Due to the vast swaths of data produced by sensors, they have revolutionized how UAV-based high-resolution imagery is evaluated for mapping of vegetation species and communities. In addition, the prevalence of new deep learning architectures and classifiers has only further widened the use-cases for CNNs in remote sensing. Some examples, include allowing for the more accurate collection/allocation of background sensor data (optical, multi-, and hyper-spectral, synthetic aperture radar, temperature and microwave radiometer, altimeter, etc.), increasing the accuracy of land use/land cover classification, image retrieval, change detection, and semantic labeling from satellite images, and allowing for the creation of new fundamental and background datasets that provide vast quantities of interdisciplinary data and background information, previously thought to be impossible [7]. Above all, the applications of CNNs to remote sensing have allowed for an overall more inclusive scientific environment, by facilitating the removal of barriers previously in place to view/access/ or request satellite imagery. These days anybody with a smart phone, internet connection and curiosity can be a citizen-scientist and things are only increasing in their openness to the general populace.

Methods

After much deliberation on whether to use 'EOBrowser' a web-based client for cloud computing of remote sensing data (Copernicus Programme), I opted to instead use 'Google Earth Engine' because of its more generally applicable framework and easy integration with Google Colab. Unfortunately, EOBrowser doesn't currently support Colab notebooks, but it is post-marked for a future release. Google Colab is a cloud computing service that allows for Python code to be written and executed in-browser for various tasks such as analysis or visualization. Google Earth Engine (GEE) is a cloud-based geospatial analysis platform that enables users to visualize and analyze satellite images of our planet [8]. GEE utilizes a in-browser scripting environment that relies on JavaScript for syntax.

The first thing to do was establish a ROI, in this case Ottawa was used. The geometry imports for the region can be viewed in script form in 'Figure 2' and visually in 'Figure 1'. Once an area was selected using the visual GUI the next step was to find a specific image, to further increase the accuracy a date filter was implemented onto the region bounds. To find an image without increasing the risk of clouds a function was implemented to the image search which measures the cloudiness of the image and if it beneath a specified threshold isn't returned, thereby guaranteeing the results returned for the time specified are the best available, the function can be seen in the first part of 'Figure 3'. The reusability of the script parameters allows for the ROI and dates to be changed easily from one location to the next. After the images are filtered by region/date and cloud level, all available results are then looped through and a real-color composite version is added to the GEE map viewer, as can be shown in 'Figure 4'. This allows for visual assessment as to the best option for the next step. After an image is chosen, the loop part of the script is commented out and the specified image then displays, also in a real-colour composite in the map viewer.

From there the short-wave-infrared and near-infrared bands are specified, before being used to calculate a built-up area index, as that is what we will be classifying in the training patches. After the building index is calculated, it too is added to the map, as shown in 'Figure 5'. To perform the supervised classification for the patches, pixels must then be sampled as both points and polygons. In ‘Figure 1’ the newly selected shape samples can be seen, with red signifying no buildings with the label ‘no_build’ and pink showing areas/points with buildings, hence the label ‘build’. Various points/polygons were selected around the region bounds to cover as many sample pixels as possible. Next using the samples specified previously as training pixels for each class, the layers are merged, and a Random Forests classifier is run over the region. After running the classifier, the displayed layer is then added to the map, as seen in ‘Figure 6’.

Now that the classification is complete, the image can be exported as patches that can then be digested by any machine learning framework. As previous, an export area or ‘bounding-box’ must first be selected, which can be shown in blue within ‘Figure 1’. The patches generated will be automatically exported as a collection, from the area to the specified folder or drive. After the box is selected, the export parameters are specified such as, patch dimensions, compression, and file name, as shown in the bottom of ‘Figure 3’. Once run, a flashing prompt will appear on the tasks pane of Google Earth Engine and must be selected before the export can complete, once done, the export will run and may take a couple of minutes to complete, before depositing the images in your Google Drive folder. Once in your google drive folder, the export patches will need to be parsed using NumPy.

Before the patches can be parsed, first the Google Colab notebook must be synced with Google Drive, as shown in the beginning of ‘Figure 7’. The drive is connected, and the appropriate file extension specified so the patches/applicable info can be imported. To retrieve the patch metadata from storage, each patch must be specified with a dictionary of the bands needed, this is done using a function and the map method to apply throughout the entire dataset. The resulting mapped data then needs to be exported, which is done using a loop and matplotlib. The resulting plots and masks can be shown near the bottom of ‘Figure 7’. From here the bands and corresponding patches are all accessible via NumPy array and can be used in any deep learning architecture as needed, or to train a custom CNN for specific patches.

Conclusion

The largest source of error would primarily be human, due to my relative inexperience with the Google Earth Engine platform. For example, errors could possibly be further diminished by using a larger variety of data sources available on Google Earth Engine, instead of limiting it to only ‘Sentinel-2’. Additionally, due to my limited experience with the GEE platform, the algorithms utilized in the script could be further optimized in addition to utilizing more of the resources available within the Google Earth Engine Platform, such as integrating some of the provided pre-made algorithms or custom datasets. The functional aspects of this project can be used to generate custom training patches from Google Earth Engine, for additional processing in any deep learning architecture. Further research is required to expand upon the methods used to scale-up the project framework to accommodate additional imagery sizes and sources. Throughout this project, my knowledge of image classification frameworks and their associated technologies has grown exponentially, and in the future, I look forward to further developing my understanding of CNNs in the confines of not just image classification, but in a more general sense as well.

Appendix.

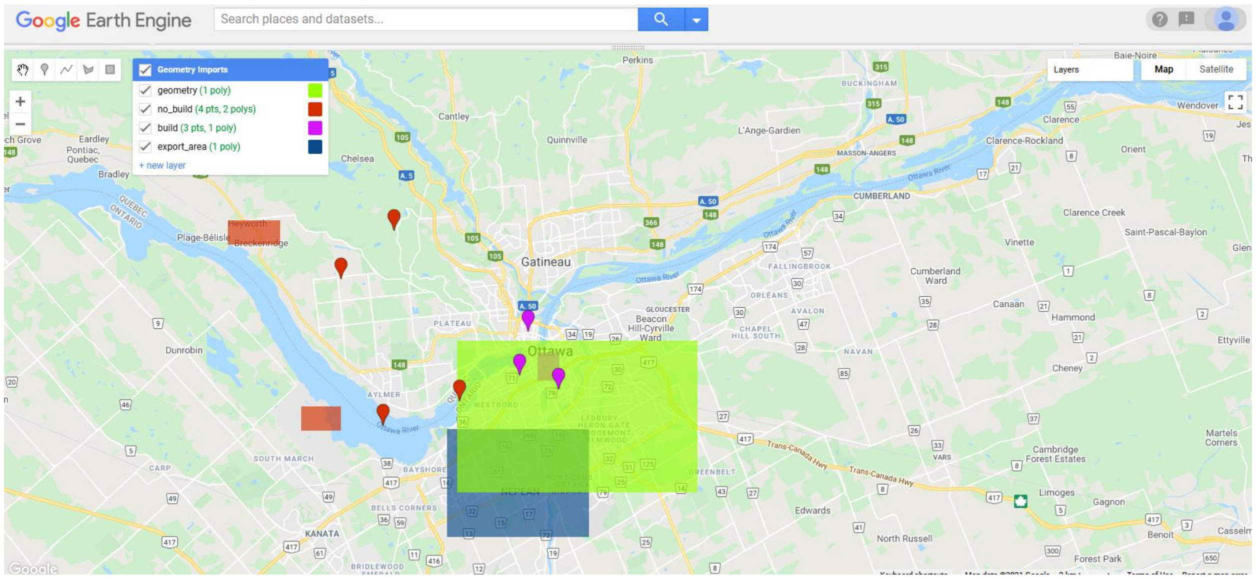
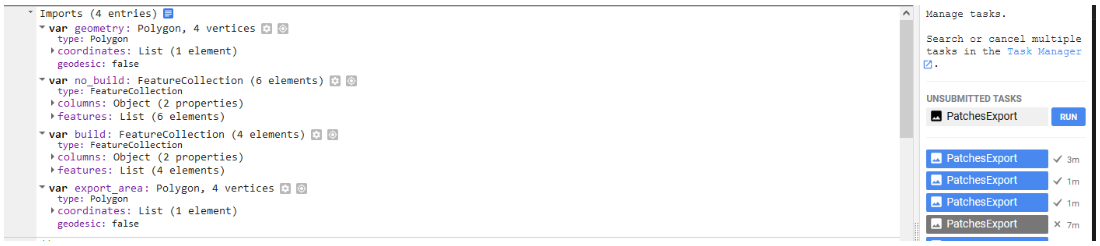


Figure 1. ROI for google earth engine and visual geometry selection. Source: Google



```
Imports (4 entries)
  var geometry: Polygon, 4 vertices
    type: Polygon
    coordinates: List (1 element)
    geodesic: false
  var no_build: FeatureCollection (6 elements)
    type: FeatureCollection
    columns: Object (2 properties)
    features: List (6 elements)
  var build: FeatureCollection (4 elements)
    type: FeatureCollection
    columns: Object (2 properties)
    features: List (4 elements)
  var export_area: Polygon, 4 vertices
    type: Polygon
    coordinates: List (1 element)
    geodesic: false
```

```
Manage tasks.

Search or cancel multiple
tasks in the Task Manager
.
```

UNSUBMITTED TASKS

PatchesExport RUN

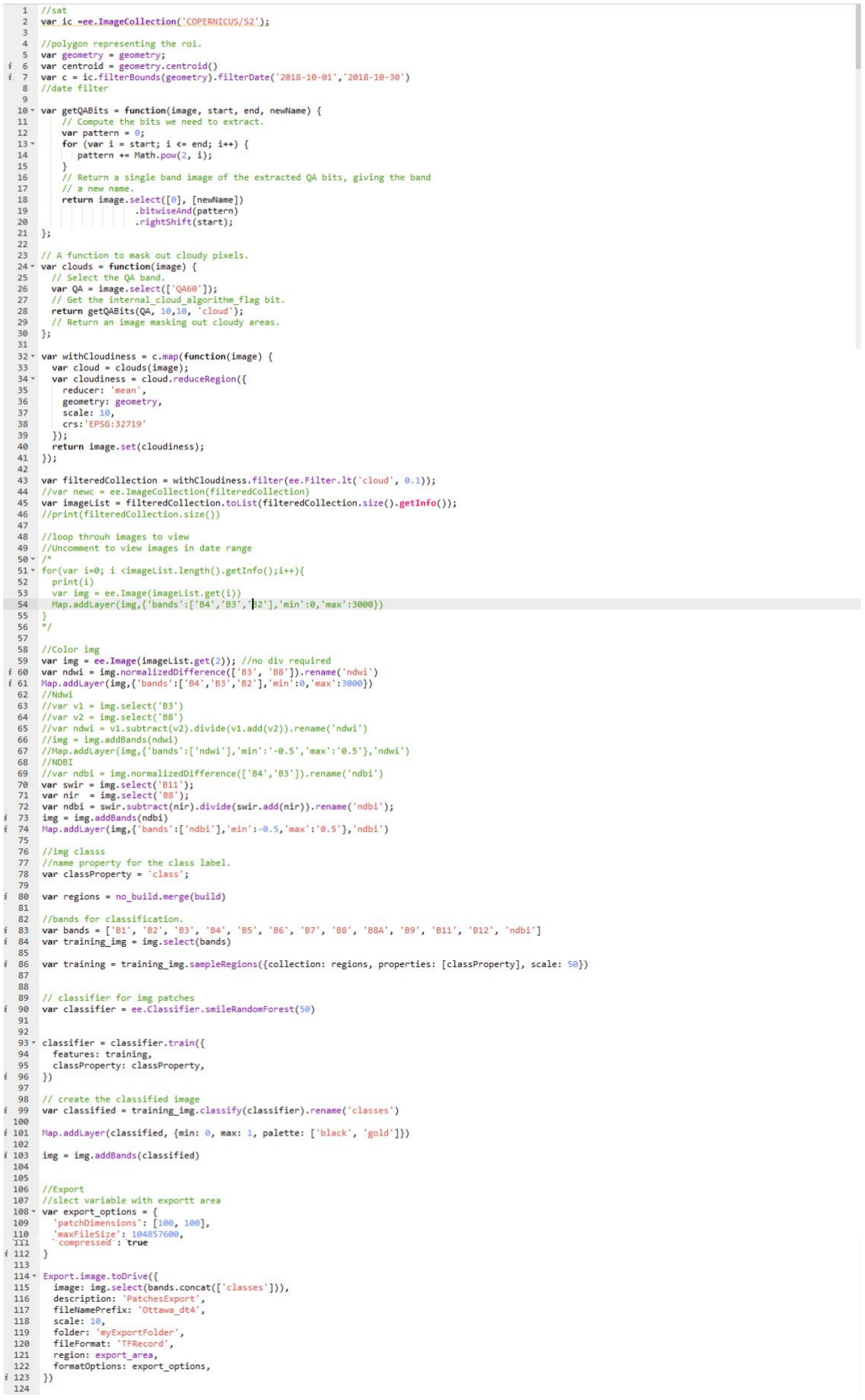
PatchesExport ✓ 3m

PatchesExport ✓ 1m

PatchesExport ✓ 1m

PatchesExport × 7m

Figure 2. Google earth engines code geometry imports. Source: Google



```javascript
//sat
var ic =ee.ImageCollection('COPERNICUS/S2');

//polygon representing the roi.
var geometry = geometry;
var centroid = geometry.centroid()
var c = ic.filterBounds(geometry).filterDate('2018-10-01','2018-10-30')
//date filter

var getQABits = function(image, start, end, newName) {
    // Compute the bits we need to extract.
    var pattern = 0;
    for (var i = start; i <= end; i++) {
       pattern += Math.pow(2, i);
    }
    // Return a single band image of the extracted QA bits, giving the band
    // a new name.
    return image.select([0], [newName])
                  .bitwiseAnd(pattern)
                  .rightShift(start);
};

// A function to mask out cloudy pixels.
var clouds = function(image) {
  // Select the QA band.
  var QA = image.select(['QA60']);
  // Get the internal_cloud_algorithm_flag bit.
  return getQABits(QA, 10,10, 'cloud');
  // Return an image masking out cloudy areas.
};

var withCloudiness = c.map(function(image) {
  var cloud = clouds(image);
  var cloudiness = cloud.reduceRegion({
    reducer: 'mean',
    geometry: geometry,
    scale: 10,
    crs:'EPSG:32719'
  });
  return image.set(cloudiness);
});

var filteredCollection = withCloudiness.filter(ee.Filter.lt('cloud', 0.1));
//var newc = ee.ImageCollection(filteredCollection)
var imageList = filteredCollection.toList(filteredCollection.size().getInfo());
//print(filteredCollection.size())

//loop throuh images to view
//Uncomment to view images in date range
/*
for(var i=0; i <imageList.length().getInfo();i++){
  print(i)
  var img = ee.Image(imageList.get(i))
  Map.addLayer(img,{'bands':['B4','B3','B2'],'min':0,'max':3000})
}
*/

//Color img
var img = ee.Image(imageList.get(2)); //no div required
var ndwi = img.normalizedDifference(['B3', 'B8']).rename('ndwi')
Map.addLayer(img,{'bands':['B4','B3','B2'],'min':0,'max':3000})
//Ndwi
//var v1 = img.select('B3')
//var v2 = img.select('B8')
//var ndwi = v1.subtract(v2).divide(v1.add(v2)).rename('ndwi')
//img = img.addBands(ndwi)
//Map.addLayer(img,{'bands':['ndwi'],'min':'-0.5','max':'0.5'},'ndwi')
//NDBI
//var ndbi = img.normalizedDifference(['B4','B3']).rename('ndbi')
var swir = img.select('B11');
var nir  = img.select('B8');
var ndbi = swir.subtract(nir).divide(swir.add(nir)).rename('ndbi');
img = img.addBands(ndbi)
Map.addLayer(img,{'bands':['ndbi'],'min':-0.5,'max':'0.5'},'ndbi')

//img classs
//name property for the class label.
var classProperty = 'class';

var regions = no_build.merge(build)

//bands for classification.
var bands = ['B1', 'B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B8', 'B8A', 'B9', 'B11', 'B12', 'ndbi']
var training_img = img.select(bands)

var training = training_img.sampleRegions({collection: regions, properties: [classProperty], scale: 50})


// classifier for img patches
var classifier = ee.Classifier.smileRandomForest(50)


classifier = classifier.train({
  features: training,
  classProperty: classProperty,
})

// create the classified image
var classified = training_img.classify(classifier).rename('classes')

Map.addLayer(classified, {min: 0, max: 1, palette: ['black', 'gold']})

img = img.addBands(classified)


//Export
//slect variable with exportt area
var export_options = {
  'patchDimensions': [100, 100],
  'maxFileSize': 104857600,
  'compressed': true
}

Export.image.toDrive({
  image: img.select(bands.concat(['classes'])),
  description: 'PatchesExport',
  fileNamePrefix: 'Ottawa_dt4',
  scale: 10,
  folder: 'myExportFolder',
  fileFormat: 'TFRecord',
  region: export_area,
  formatOptions: export_options,
})
```

Figure 3. Google Earth Engine script for imagery. Source: Google

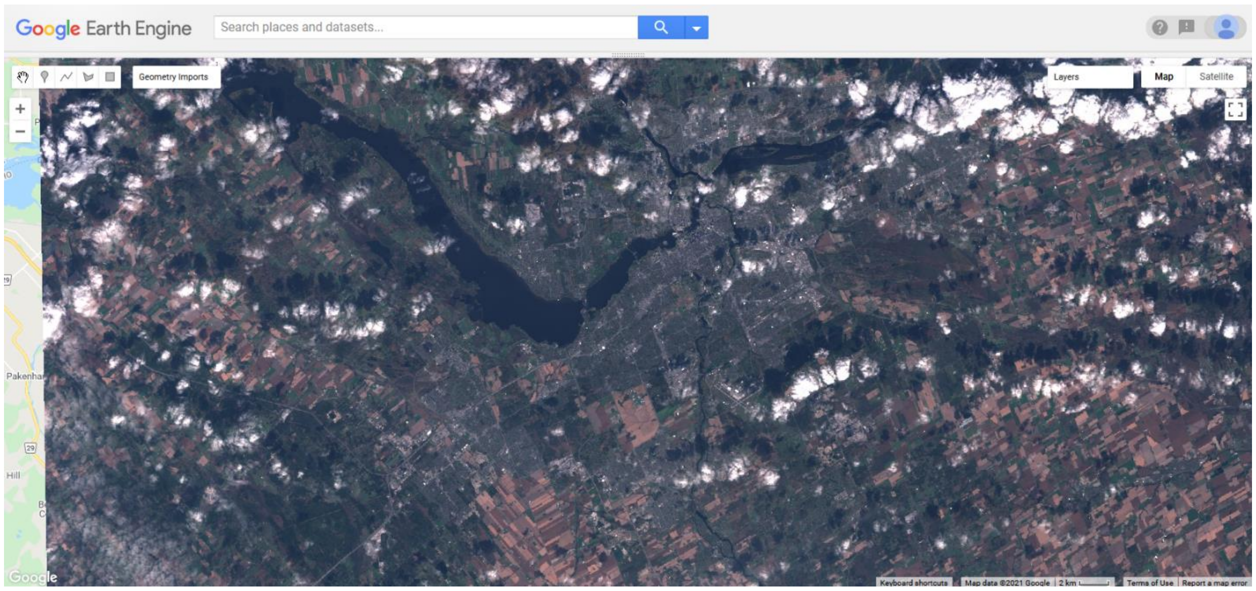


Figure 4. Real colour composite layer in Google Earth Engine. Source: Google

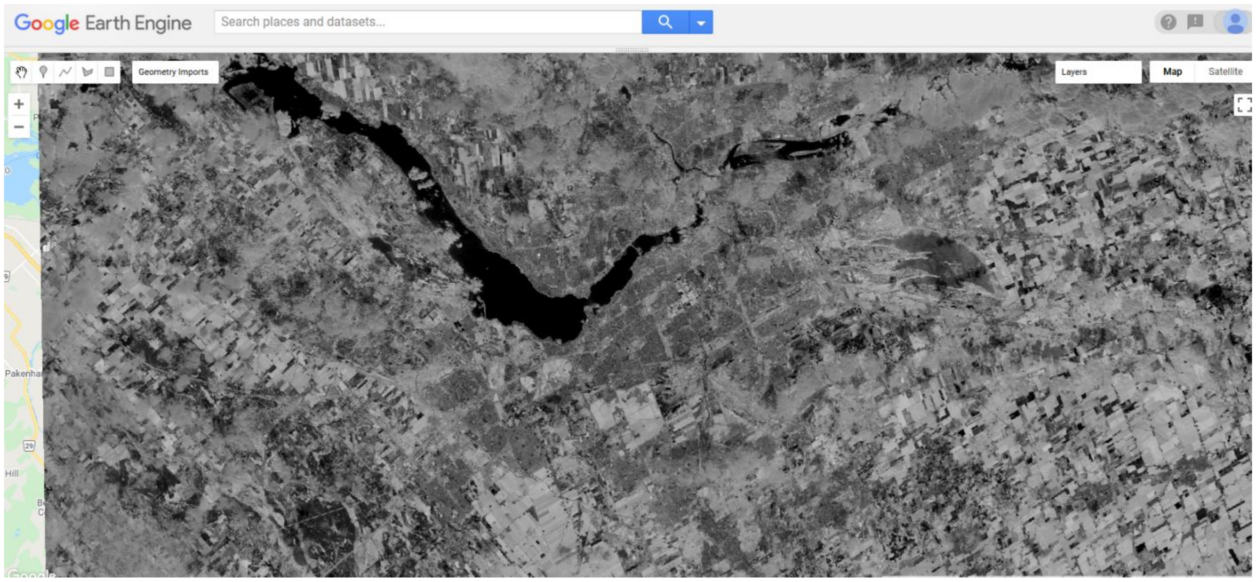


Figure 5. Built-up area index layer in Google Earth Engine. Source: Google

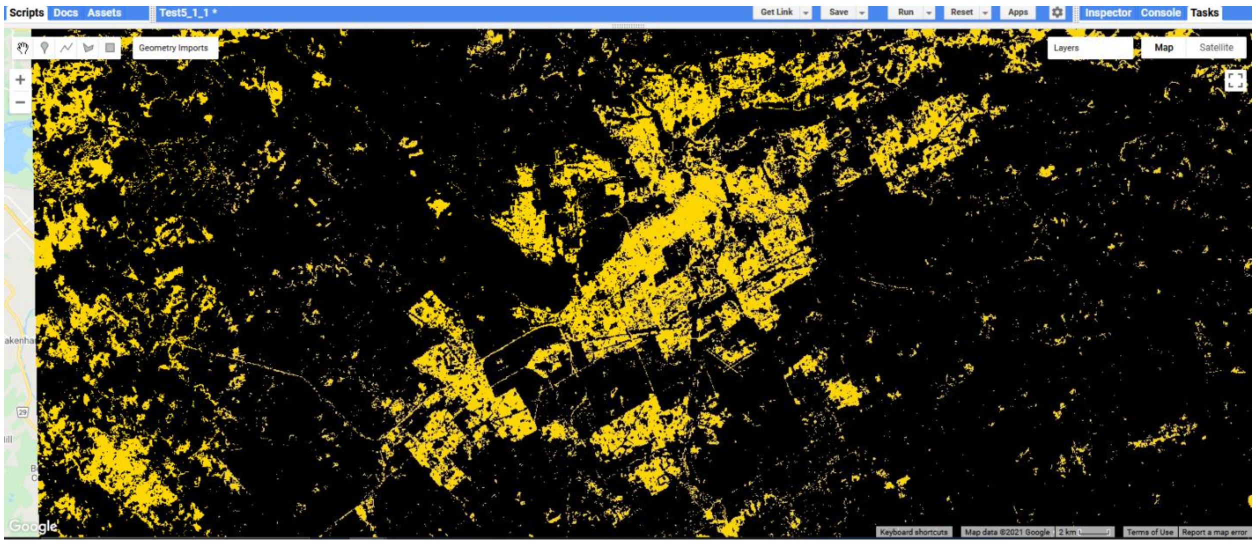
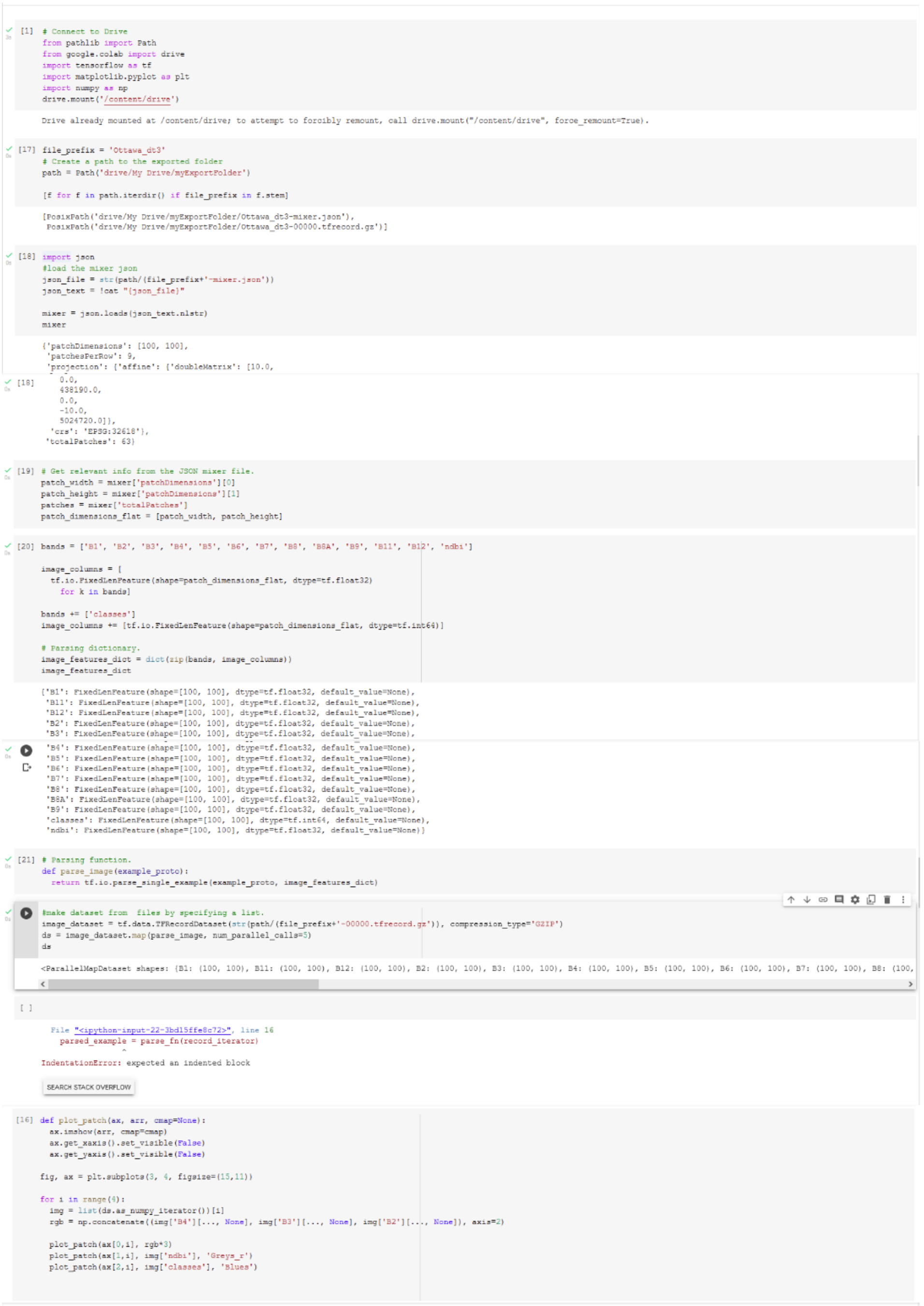


Figure 6. Classified layer (yellow = buildings and black = no buildings) in Google Earth Engine. Source: Google



```
[1] # Connect to Drive
    from pathlib import Path
    from google.colab import drive
    import tensorflow as tf
    import matplotlib.pyplot as plt
    import numpy as np
    drive.mount('/content/drive')

    Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
[17] file_prefix = 'Ottawa_dt3'
     # Create a path to the exported folder
     path = Path('drive/My Drive/myExportFolder')

     [f for f in path.iterdir() if file_prefix in f.stem]

     [PosixPath('drive/My Drive/myExportFolder/Ottawa_dt3-mixer.json'),
      PosixPath('drive/My Drive/myExportFolder/Ottawa_dt3-00000.tfrecord.gz')]
```

```
[18] import json
     #load the mixer json
     json_file = str(path/(file_prefix+'-mixer.json'))
     json_text = !cat "{json_file}"

     mixer = json.loads(json_text.nlstr)
     mixer

     {'patchDimensions': [100, 100],
      'patchesPerRow': 9,
      'projection': {'affine': {'doubleMatrix': [10.0,
```

```
[18]    0.0,
        438190.0,
        0.0,
        -10.0,
        5024720.0]},
      'crs': 'EPSG:32618'},
     'totalPatches': 63}
```

```
[19] # Get relevant info from the JSON mixer file.
     patch_width = mixer['patchDimensions'][0]
     patch_height = mixer['patchDimensions'][1]
     patches = mixer['totalPatches']
     patch_dimensions_flat = [patch_width, patch_height]
```

```
[20] bands = ['B1', 'B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B8', 'B8A', 'B9', 'B11', 'B12', 'ndbi']

     image_columns = [
       tf.io.FixedLenFeature(shape=patch_dimensions_flat, dtype=tf.float32)
         for k in bands]

     bands += ['classes']
     image_columns += [tf.io.FixedLenFeature(shape=patch_dimensions_flat, dtype=tf.int64)]

     # Parsing dictionary.
     image_features_dict = dict(zip(bands, image_columns))
     image_features_dict

     {'B1': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B11': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B12': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B2': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B3': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B4': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B5': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B6': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B7': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B8': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B8A': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'B9': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None),
      'classes': FixedLenFeature(shape=[100, 100], dtype=tf.int64, default_value=None),
      'ndbi': FixedLenFeature(shape=[100, 100], dtype=tf.float32, default_value=None)}
```

```
[21] # Parsing function.
     def parse_image(example_proto):
       return tf.io.parse_single_example(example_proto, image_features_dict)
```

```
#make dataset from  files by specifying a list.
image_dataset = tf.data.TFRecordDataset(str(path/(file_prefix+'-00000.tfrecord.gz')), compression_type='GZIP')
ds = image_dataset.map(parse_image, num_parallel_calls=5)
ds

<ParallelMapDataset shapes: {B1: (100, 100), B11: (100, 100), B12: (100, 100), B2: (100, 100), B3: (100, 100), B4: (100, 100), B5: (100, 100), B6: (100, 100), B7: (100, 100), B8: (100,
```

```
[ ]

  File "<ipython-input-22-3bd15ffe8c72>", line 16
    parsed_example = parse_fn(record_iterator)
                   ^
IndentationError: expected an indented block
```

SEARCH STACK OVERFLOW

```
[16] def plot_patch(ax, arr, cmap=None):
       ax.imshow(arr, cmap=cmap)
       ax.get_xaxis().set_visible(False)
       ax.get_yaxis().set_visible(False)

     fig, ax = plt.subplots(3, 4, figsize=(15,11))

     for i in range(4):
       img = list(ds.as_numpy_iterator())[i]
       rgb = np.concatenate((img['B4'][..., None], img['B3'][..., None], img['B2'][..., None]), axis=2)

       plot_patch(ax[0,i], rgb*3)
       plot_patch(ax[1,i], img['ndbi'], 'Greys_r')
       plot_patch(ax[2,i], img['classes'], 'Blues')
```

[16]

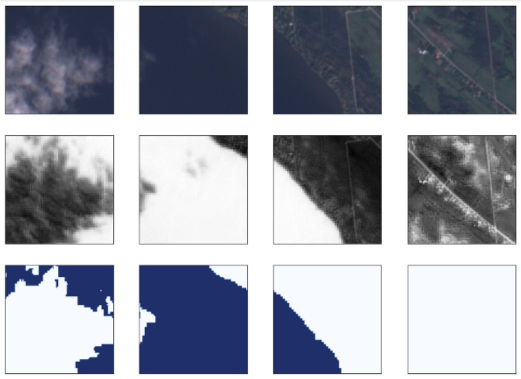


Figure 7. Google Collab notebook for processing. Source: Google

Works Citied.

[1] W. Rawat and Z. Wang, "Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review," Neural Computation, vol. 29, no. 9, pp. 2352–2449, Sep. 2017, doi: 10.1162/neco_a_00990.

[2] Stanford, "CS231n Convolutional Neural Networks for Visual Recognition." https://cs231n.github.io/ (accessed Nov. 24, 2021).

[3] Z.-Q. Zhao, P. Zheng, S. Xu, and X. Wu, "Object Detection with Deep Learning: A Review," arXiv:1807.05511 [cs], Apr. 2019, Accessed: Nov. 28, 2021. [Online]. Available: http://arxiv.org/abs/1807.05511

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," Commun. ACM, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.

[5] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," arXiv:1311.2901 [cs], Nov. 2013, Accessed: Nov. 29, 2021. [Online]. Available: http://arxiv.org/abs/1311.2901

[6] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," arXiv:1206.5533 [cs], Sep. 2012, Accessed: Dec. 01, 2021. [Online]. Available: http://arxiv.org/abs/1206.5533

[7] A. K. Cherian and E. Poovammal, "Classification of remote sensing images using CNN," IOP Conf. Ser.: Mater. Sci. Eng., vol. 1130, no. 1, p. 012084, Apr. 2021, doi: 10.1088/1757-899X/1130/1/012084.

[8] N. Gorelick, M. Hancher, M. Dixon, S. Ilyushchenko, D. Thau, and R. Moore, "Google Earth Engine: Planetary-scale geospatial analysis for everyone," Remote Sensing of Environment, vol. 202, pp. 18–27, Dec. 2017, doi: 10.1016/j.rse.2017.06.031.

[9] N. Ståhl and L. Weimann, "Identifying Wetland Areas in Historical Maps using Deep Convolutional Neural Networks," arXiv:2108.04107 [cs], Aug. 2021, Accessed: Dec. 04, 2021. [Online]. Available: http://arxiv.org/abs/2108.04107

[10] J. Dietrich, CNN-Supervised Classification. 2021. Accessed: Dec. 04, 2021. [Online]. Available: https://github.com/geojames/CNN-Supervised-Classification

[11] "Remote Sensing | Free Full-Text | Comparing Deep Learning and Shallow Learning for Large-Scale Wetland Classification in Alberta, Canada | HTML." https://www.mdpi.com/2072-4292/12/1/2/htm (accessed Dec. 07, 2021).